Sections and Chapters

Gubert Farnsworth

Ngày 16 tháng 12 năm 2020

STRING MATCHING

1 String Matching Problem

Tìm kiếm chuỗi hiện diện rất nhiều trong cuộc sống, ví dụ như: tìm tên thầy dạy DSA trong danh sách giảng viên, tìm tên trong bảng điểm,.. hoặc trong khoa học, như là tìm liệu cấu trúc ADN của virus này có trong virus khác hay không.

Trong Tin học, các trình soạn thảo văn bản thường phải tìm kiếm (tất cả) các lần xuất hiện của một đoạn văn bản trong một văn bản dài. Thông thường văn bản được chỉnh sửa liên tục, và các phần văn bản cần tìm kiếm thì được nhập bởi người dùng. Việc phát minh ra các thuật toán tìm kiếm chuỗi hiệu quả đã giúp ích rất nhiều cho các bài toán kể trên.

Bài toán tìm kiếm chuỗi thường được mô tả theo mô hình sau:
Một chuỗi cần tìm kiếm S có dạng một chuỗi kí tự S[1..m], cần tìm chuỗi S trong một đoạn văn bản T[1..n], với m <= n. Đồng thời, tất cả các ký tự trong T và S đều thuộc về một tập hữu hạn các ký tự cho trước. Chuỗi S được gọi là xuất hiện bắt đầu từ vị trí p + 1 nếu thoả điều kiện: T[p + j] = S[j], với 1 <= j <= m.

Có 3 thuật toán thường được dùng để tìm kiếm chuỗi:

- Thuật toán Brute-force (vét cạn, hay còn gọi là thuật trâu)
- Rabin - Karp
- Knuth - Morris - Pratt

Chi tiết của từng thuật toán sẽ được trình bày bên dưới.

2 String Matching Algorithms

1. Brute-force

   Giải thuật Brute-Force, hay còn gọi là vét cạn, là thuật toán đơn giản nhất trong các thuật toán tìm kiếm chuỗi con *pattern* trong chuỗi cha *text*.

   Có thể giải thích đơn giản, giải thuật Brute-Force so sánh lần lượt mỗi chuỗi con *subtext* của *text* có cùng chiều dài với *pattern* với *pattern*, nếu tìm được, trả về kết quả là vị trí được tìm thấy; khi không tìm được kết quả mong muốn, trả về giá trị quy ước là không tìm thấy.

   Trong ví dụ sau, ta sẽ làm rõ cách hoạt động của giải thuật này:
   text = `Let them go!`
   pattern = `them`

Let them go!
them

Let them go!
 them

Let them go!
  them

```
Let them go!
  them
```

```
Let them go!
   them
```

```
Let them go!
    them
```

```
Let them go!
    them
```

```
Let them go!
    them
```

```
Let them go!
    them
```

Ta tìm thấy chuỗi pattern tại vị trí thứ 4!

Từ ví dụ trên, ta thiết kế mã giả cho giải thuật Brute-Force:

```
function: Brute-Force String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
subtext: 1 đoạn thân của chuỗi cha có cùng chiều dài với pattern.
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  vị trí tìm thấy = -1
  subtext = chuỗi con đầu text có độ dài bằng pattern
  while (chưa tìm thấy hoặc chưa tới cuối text)
    if (từng ký tự của subtext = pattern):
      trả về vị trí
    else:
      dịch chuyển chuỗi con subtext trong text sang phải 1 chữ cái
  Trả về: vị trí tìm thấy
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N
Phân tích độ phức tạp của thuật toán trong trường hợp xấu nhất:

- Mỗi lần so sánh với *subtext*, *pattern* phải so sánh nhiều nhất là M lần (trong trường hợp cả M - 1 ký tự đầu đều đúng).
- Có tất cả N - M + 1 chuỗi, vậy số chuỗi cần so sánh nhiều nhất là N - M + 1 *subtext* như vậy (trong trường hợp N - M + 2 chuỗi *subtext* đầu không trùng với *pattern*)
  → Cần M(N - M + 1) lần. Vì duyệt tới cuối mảng nên đây là trường hợp tìm thấy ở cuối mảng, hoặc không tìm thấy
  → Cận trên $O(MN)$ (vì N $>$ N - M + 1).
  → Cấp phát bộ nhớ: 0.

Phân tích độ phức tạp của thuật toán trong trường hợp tốt nhất:

- Trong trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*.
- Như vậy, chỉ cần tốn M lần so sánh các ký tự.
  → Cần M lần.
  → Cận trên $O(M)$.
  → Cấp phát bộ nhớ: 0.

Độ phức tạp của thuật toán trong trường hợp trung bình: $O(MN)$

Đánh giá:

- Dễ hiểu, thuật toán này chỉ duyệt từ đầu đến cuối, so sánh tuần tự từng chuỗi con với chuỗi cần tìm kiếm.
- Không cần bước tiền xử lý (như các thuật toán được trình bày bên dưới).

- Độ phức tạp $O(MN)$. Không cần xin thêm bộ nhớ.

2. Rabin-Karp

3. Knuth-Morris-Pratt

   Giải thuật Knuth-Morris-Pratt, về cơ bản cũng giống như thuật toán Brute-Force, tuy nhiên, chỉ khác ở chỗ, Brute-Forch khi so sánh *pattern* và *subtext*, Brute-Force so sánh toàn bộ các ký tự của chúng lại từ đầu, còn Knuth-Morris-Pratt, từ lần so sánh trước, sẽ quyết định có so sánh pattern với *subtext* kế tiếp không, và nếu không sẽ nhảy bao nhiêu bước đến *subtext* khác. (đã biết được chúng giống nhau), từ đó tiết kiệm được chi phí giải bài toán.

   Để dễ hiểu, ta xét 1 ví dụ như sau:

   text = ababcababd
   pattern = ababd

---

ababcababd
ababd // So sánh như Brute-Force

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd

---

ababcababd
ababd // ta thấy ký tự thứ 5 không đúng, nên ta sẽ tìm chuỗi con tiếp theo để so sánh.
// vả lại ta/máy tính cũng thấy rằng có 4 ký tự đầu của pattern trùng với subtext
// Trong 4 ký tự đó (abab), có có **2 ký tự đầu** giống **2 ký tự cuối**
// Như vậy, lần tiếp theo ta nên so sánh pattern với chuỗi ab**abcab**abd

---

ababcabdabd
ababd // Để ý rằng ta sẽ so sánh với chuỗi ab**abcab**abd (nhảy 2 bước so với hiện tại) mà
// không phải là a**babca**babd, vì subtext của bước so sánh này luôn không trùng
// pattern, giả sử điều ngược lại, pattern[1, 4] = text[2, 6] vì chuỗi dài nhất để chuỗi
// đầu pattern[1, 4] và chuỗi cuối pattern[2, 4] giống nhau là 2 (ab, lớn nhất),
// Lưu ý rằng ta lấy pattern[1, 4] để tìm chuỗi chung là vì nó là phần đã được so sánh
// đúng ở lượt trước, và ta có nhu cầu tái sử dụng các phép so sánh đúng này!
// Ngoài ra, chuỗi cuối ta chỉ lấy của pattern[2, 4] vì bước tiếp theo ta phải nhảy
// sang phải (nếu lấy pattern[1, 4] mà cả 4 ký tự giống nhau thì bước kế tiếp
// ta nhảy về vị trí đang đứng, ở giải thuật này)
// Ta có: pattern[1, 3] = text[2, 4], mà text[2, 4] = pattern[2, 4] (lượt so sánh trước)
// như vậy là: pattern[1, 3] = pattern[2, 4], tức là có chuỗi độ dài 3 vừa là chuỗi đầu
// vừa là chuỗi cuối của pattern[1, 4] (vô lý vì 2 là số lớn nhất).

---

ababcababd
ababd // Ta so sánh ký tự thứ 3 trở đi (vì ta/máy tính biết đã có 2 ký tự đầu tiên là
// giống nhau rồi!), ở bước này, phép so sánh là sai.
// Vị trí sai là 3, ta thấy không có chuỗi chung vừa là
// chuỗi đầu và vừa là chuỗi cuối của pattern[1, 2], nên ta nhảy 1 bước sang phải.

---

ababcababd
ababd

---

ababcababd
ababd // Tiếp tục so sánh như Brute-Force

---

ababcababd
ababd

```
ababcababd
    ababd
```

```
ababcababd
    ababd
```

```
ababcababd
    ababd
```

```
ababcababd
    ababd       // Ok, tìm thấy rồi!
```

Note: Với string là chuỗi có độ dài n. và j là số nguyên dương $0 < j < n + 1$
Chuỗi đầu: Prefix, có dạng prefix[] = string[1, j], chuỗi cuối: Suffix, có dạng Suffix[] = string[j, n].

Từ ví dụ trên, ta tìm cách thiết kế ý tưởng của KMP để tìm chuỗi 1 cách tổng quát:

- Ta thấy rằng, để KMP tìm cách nhảy hiệu quả, cần phải xử lý trước pattern, tìm chuỗi chung vừa là chuỗi đầu, vừa là chuỗi cuối của pattern. Bước này gọi là bước tìm Failure Function, tại mỗi vị trí j trong pattern ($0 < j < \text{len(pattern)} + 1$), tìm độ dài lớn nhất của chuỗi chung và lưu vào mảng $f[]$ để lưu trữ.
- Khi đang so sánh pattern và subtext, ta có thể gặp các trường hợp ($0 < i <= \text{len(subtext)}$):
  1. Nếu pattern[i] = subtext[i], tăng thêm i để so sánh ký tự kế tiếp.
  2. Nếu pattern[i] $\neq$ subtext[i], và i = 1, tức là ký tự đầu của chúng khác nhau, đơn giản là chỉ cần nhảy subtext sang bên phải 1 bước.
  3. Nếu pattern[i] $\neq$ subtext[i], và $i > 1$, tức là đã có 1 lượng i - 1 ký tự ở trước đã được so sánh đúng, bây giờ là đến lúc cần dùng mảng f đã tính trước ở bên trên, ta lấy f[i - 1] (chính là độ dài lớn nhất của chuỗi chung của pattern[1, i - 1]) và ta sẽ nhảy subtext 1 lượng min((i - 1) - f[i - 1], 1) để chuẩn bị cho lượt so sánh kế tiếp.

Từ ý tưởng trên, ta xây dựng mã giả cho thuật toán KMP:

```
function: Build Failure Function
pattern: chuỗi cần tìm
Trả về: Mảng f lưu độ dài lớn nhất f[i] của chuỗi chung prefix và suffix
   của pattern[1, i].

  f[] = 0 để lưu mảng trả về
  fpre = 0;   // lưu f[i - 1] trong vòng lặp bên dưới
  i = 2;    // Ta chỉ tính f[2] trở đi, f[1] luôn bằng 0
  while (i <= n)
    Nếu pattern[i] == pattern[f[i - 1] + 1]
      fpre++
      f[i] = fpre // tận dụng kết i - 1 kết quả so sánh của f[i - 1], ta
         chỉ cần so sánh 1 ký tự.
      i++
    Nếu pattern[i] != pattern[f[i - 1] + 1] và f[i - 1] = 0
      // Tức là suffix và prefix là chuỗi có độ dài f[i - 1] + 1 có ký
         tự cuối không giống
      // -> f[i] = 0
      f[i] = 0
      i++
    Nếu pattern[i] != pattern[f[i - 1] + 1] và f[i - 1] > 0
      // Trường hợp này, có thể f[i] = f[i - 1]
      // Ta sẽ tận dụng pattern[1, f[i- 1]] = pattern[i - f[i-1], i]
      // Và f[f[j]] (độ dài này lớn nhất là j - 1)
      // Ví dụ AAABAAAA khi xét i = 8, ta lợi dụng chuỗi [AA]ABAAAA =
         A[AA]BAAAA và [AAA]BAAAA = AAAB[AAA]A
      // Suy ra được [AA]ABAAAA = A[AA]BAAAA = AAABA[AA]A
      // Như thế chỉ cần xét ký tự AA[A]BAAAA và AAABAAA[A] để xác định
         f[i]
      fpre = f[fpre]
  Trả về: f[]
```

```
27        function: KMP String Matching
28        text: chuỗi cha
29        pattern: chuỗi cần tìm
30        Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.
31
32          i = 1; j = 1; (i đánh dấu vị trí đang duyệt trong text, j đánh dấu vị
               trí đang duyệt trong pattern)
33          while (i < len(text) - len(pattern))
34            Nếu pattern[j] == text[i]
35              Tiếp tục i++ và j++ và so sánh
36                Nếu j = len(pattern) trả về đã tìm thấy pattern tại vị trí i - j.
37            Nếu pattern[j] != text[i] và j == 1
38              i = i + 1  //Chuyển nhảy subtext sang phải 1 đơn vị
39            Nếu pattern[j] != text[i] và j > 1
40              j = f[i] và tiếp tục so sánh text[i] và pattern[j] // sau khi gán
                   j = f[i] (giảm j) thì ta đã cố tình dịch chuyển subtext đang
                   so sánh sang subtext khác!
41          Nếu trong vòng lặp không tìm thấy pattern, ta đã duyệt hết text, trả
               về -1 báo kết quả không tìm thấy pattern trong text.
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N
Phân tích độ phức tạp của thuật toán trong trường hợp xấu nhất:

- Trong quá trình tiền xử lý (tính hàm Failure Function), trong vòng while chỉ tốn nhiều nhất 1 - 2 lần để tính f[i], vậy độ phức tạp là $O(M)$
- Trong quá trình so sánh, mỗi lần, ta hoặc sẽ tăng i lên 1 đơn vị, hoặc sẽ giảm j ít nhất 1 đơn vị (trong trường hợp dịch j), nghĩa là tăng k lên 1 đơn vị, k = i - j < N. Vậy tổng cộng trong vòng lặp có thể lặp < 2n lần (N lần tăng i, ít hơn N lần tăng k), như vậy độ phức tạp là $O(2N)$.
  → Độ phức tạp $O(M + N)$.
  → Cấp phát bộ nhớ: $O(M)$.

Phân tích độ phức tạp của thuật toán trong trường hợp tốt nhất:

- Trong trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*.
- Như vậy, chỉ cần tốn M lần so sánh các ký tự. Ngoài ra còn tốn thêm chi phí tính Failure Function $O(M)$.
  → Cận trên $O(M)$.
  → Cấp phát bộ nhớ: $O(M)$.

Độ phức tạp của thuật toán trong trường hợp trung bình:

- Cận trên $O(N + M)$.
- Cấp phát bộ nhớ: $O(M)$.

Đánh giá:

- Nhanh, chi phí tuyến tính.
- Phải xử lý trước khi tìm kiếm thực sự.
- Độ phức tạp $O(M + N)$. Chi phí bộ nhớ $O(M)$.

PROGRAMING

(abc)

1 Introduce

2 Example Test